Module3: Exception Handling and Multithreading

Introduction to Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error.
- An exception is an event that disrupts the normal flow of the program.
- In Java Exception is an object which is thrown at run time.
- The exceptions can be

-Division by zero
-Array out of bound access exception
-End of file access etc.

Exception Handling:

“Exception handling is the mechanism to handle run time errors, so that the normal flow of application can be maintained.”

Java’s exception handling brings run-time error management into the object-oriented world.

Java Exception Handling is managed by using 5 Keywords.

1. try
2. catch
3. throw
4. throws
5. finally

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

ExceptionType is the type of exception that has occurred.

1. Program Statements that we monitor for exceptions are contained in try block.

2. If an exception occurs in try block, it is thrown. The code can catch this exception using catch block and handle the exception in a rational manner.

▯ System generated exceptions are automatically thrown by the Java runtime.

▯ Catch block is used to handle the exception, called Exception Handler.

▯ Must be used after try block only. We can use multiple catch blocks with a single try block.

3. The throw keyword is used to manually throw an exception

4. Any exception that is thrown out of a method be specified using throws clause.

5. Any code that absolutely must be executed after a try block completes is put in finally block.

Uncaught Exceptions

When we don’t handle an exception in a Java Program

```
class Excep1
{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.

-This causes the execution to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.

-If no exception handlers are defined, the exception is caught by the default handler provided by the Java run-time system.

-The default handler displays a string describing the exception, prints a

 The stack trace will always show the sequence of method invocations that led up to the error.

```
public class Excep2
{
static void find()
{
int d = 0;
int a = 10 / d;
}
public static void main(String args[])
{
find();
}
}
```

```
D:\OOP\excep>javac Excep2.java

D:\OOP\excep>java Excep2
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Excep2.find(Excep2.java:6)
        at Excep2.main(Excep2.java:10)
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

-The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

1. Exception Handling Fundamentals

Exception –

– Exception is an abnormal condition that arises in a code sequence at run time.

– An exception is a runtime error.

– An exception is an event that disrupts the normal flow of the program.

– It is an object which is thrown at runti

Eg: The Exceptions can be division by zero, array out of bound access, end of file access, ... etc.

Exception Handling –

In Java, Exception handling is the mechanism to handle runtime errors, so that the normal flow of application can be maintained.

Java Exception handling is managed by using five keywords:

try, catch, throw, throws & finally.

The general form of exception-handling block

```
try {
    // block of code to monitor for errors
}
catch ( ExceptionType1 obj)
{
    // exception handler for
                    ExceptionType 1
}
catch ( ExceptionType2 obj)
{   // exception handler for
                    ExceptionType2
}
- - - -
- - - -
finally {
    // block of code to be executed
       after try block ends
}
```

ExceptionType — is Type of exception that occured.

① Program statements that we monitor for exceptions are contained in try block.

② If an exception occurs in try block, it is thrown. your code can catch this exception using catch and handle it in a some rational manner.

- System generated exceptions are automatically thrown by the Java runtime.
- The catch block is used to handle the exception. (Exception handler).
- Must be used after try block only.
- We can use multiple catch blocks with a single try.

③ The throw keyword is used to manually throw an exception.

④ Any exception that is thrown out of a method must be specified as such by a throws clause.

⑤ Any code that absolutely must be executed after a try block completes is put in finally block.

Page No.

Problem without Exception Handling

```
public class ExcepDemo1
{
    public static void main(String args[])
    {   int y = 50;
        int x = y/0;  // May throw Exception
        System.out.println("Rest of code");
        System.out.println("Welcome");
    }
}
```

O/p — Exception in Thread main :
java.lang.ArithmeticException : / by zero
The rest of the code is not executed.

Solution by using Exception Handling

```
public class ExcepDemo2
{
    public static void main (String args[])
    {
        try {
            int y = 50;
            int x = y/0;
        }
        catch (ArithmeticException e)
        {
            System.out.println (e);
            Sop (" should not be zero in
                  denominator ");
        }
        Sop (" Rest of code");
        Sop (" welcome to Exception Handling");
    }
}
```

O/P: java.lang.ArithmeticException : / by zero
DivisionError — should not be zero in denominator
Rest of code
welcome to Exception Handling

Common Scenarios Where Exceptions occur

① ArithmeticException
— Any number divided by zero
Eg: int a = 50/0 ; // ArithmeticException

② NullPointerException
— Any operation performed on null variable
Eg: String str = null;
x = str.length() ; //NullPointerException

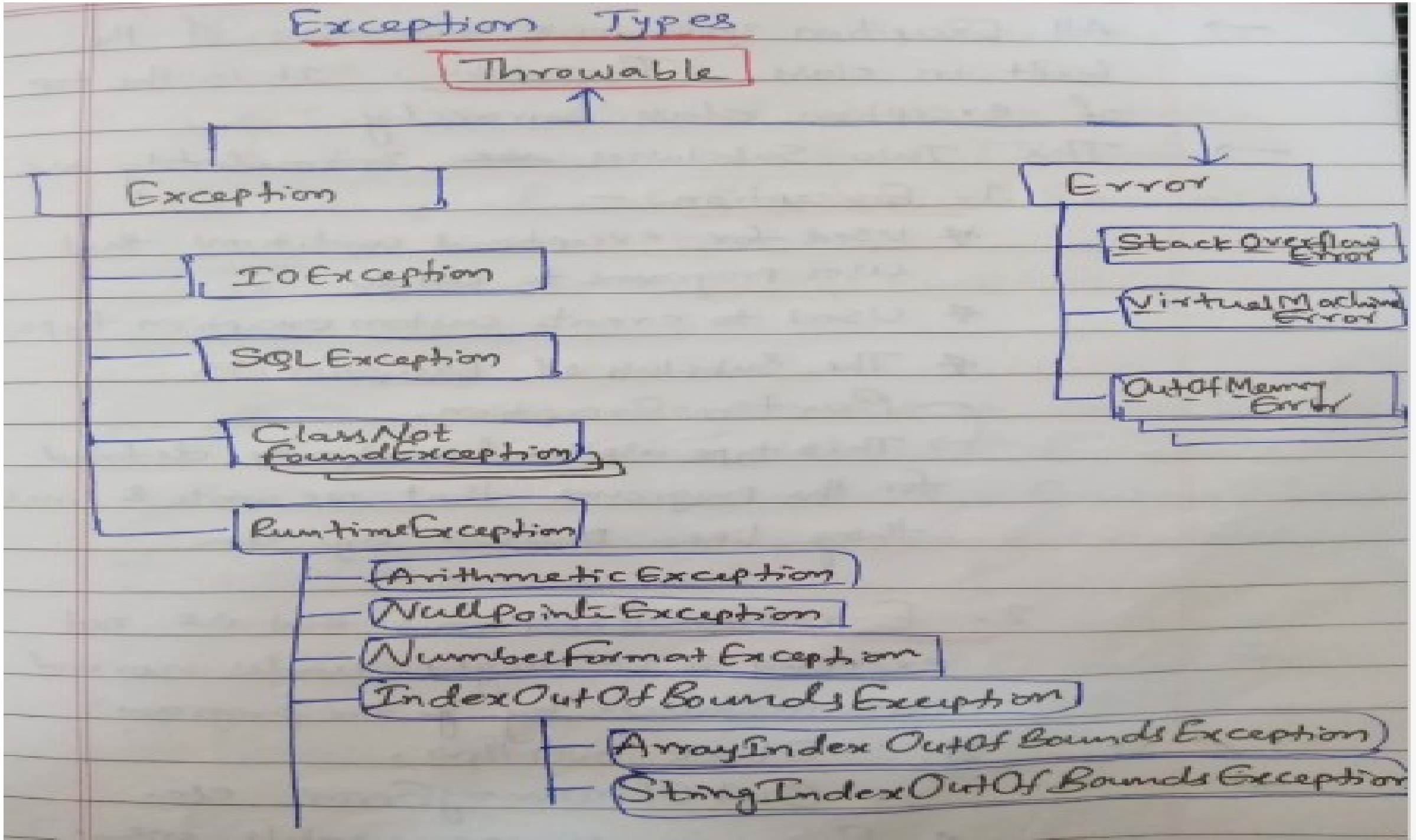
③ NumberformatException
— Converting a string into numbers
Eg: Strng str = "abc";
int i = Integer.parseInt(str);
// NumberformatException

④ ArrayIndexOutOfBoundsException
— Accessing an array beyond its capacity
Eg: int a[] = new int[5];
a[5] = 20; // Exception i.e.

↓

ArrayIndexOutOfBoundsException.

Exception Types



→ All Exception Types are subclasses of the built in class Throwable. It is the top of exception class heirarchy.

→ The Two Subclasses immediately are

1. Exception —
   - Used for exceptional conditions that user programs to catch
   - Used to create custom exception types.
   - The Subclass of Exception is
     ↳ RuntimeException.
     ↳ This type are automatically defined for the programs that we write & includ thing like Division by zero.

2. Error — Exceptions that are not expected to be caught under normal circumstances by the program.
   Eg: Stack overflow,
   OutOfMemoryError etc.
   - Error is unrecoverable one.

The two categories of Exceptions are

1. Checked Exception
   - These are checked at compile time.
   - The classes that extend Throwable except RuntimeException & Error are the checked exceptions

   Eg: IOException, SQLException.

2. Unchecked Exception
   - There are not checked at compile time, rather they are checked at ~~compile~~ Runtime time
   - The classes that extend RuntimeException are known as unchecked exceptions.

   Eg: ArithmeticException,
   NullPointerException,
   ArrayIndexOutOfBoundsException etc.

2. UnCaught Exceptions

- When Java Run-Time System detects the exception, it constructs a new exception object and then throws this exception.
- If no exception handler is provided, the exception is caught by the default handler provided by the Java Runtime System.
- The default handler display a String describing the exception, prints a stack trace.

Eg: ③

```
public class ExcepDemo3
{
    public static void main (String args[])
    {
        int d = 25;
        int c = d/0;
    }
}
```

↳ o/p:

StackTrace →

Eg: ②

```
class ExcepDemo {
    static void method()
    {
        int d = 0;
        int a = 10/d;
    }
    public static void main (String args[])
    {
        ExcepDemo.method();
    }
}
```

↳ o/p

StackTrace :

↳ Point at which exception occured, and terminates the program.

3. Using try, catch, throw, throws & finally

(1) Using try and catch

Handling an exception has 2 Advantages.

① Allows to fix errors.

② Prevents the program from abnormal termination.

To guard against and handle a runtime error,

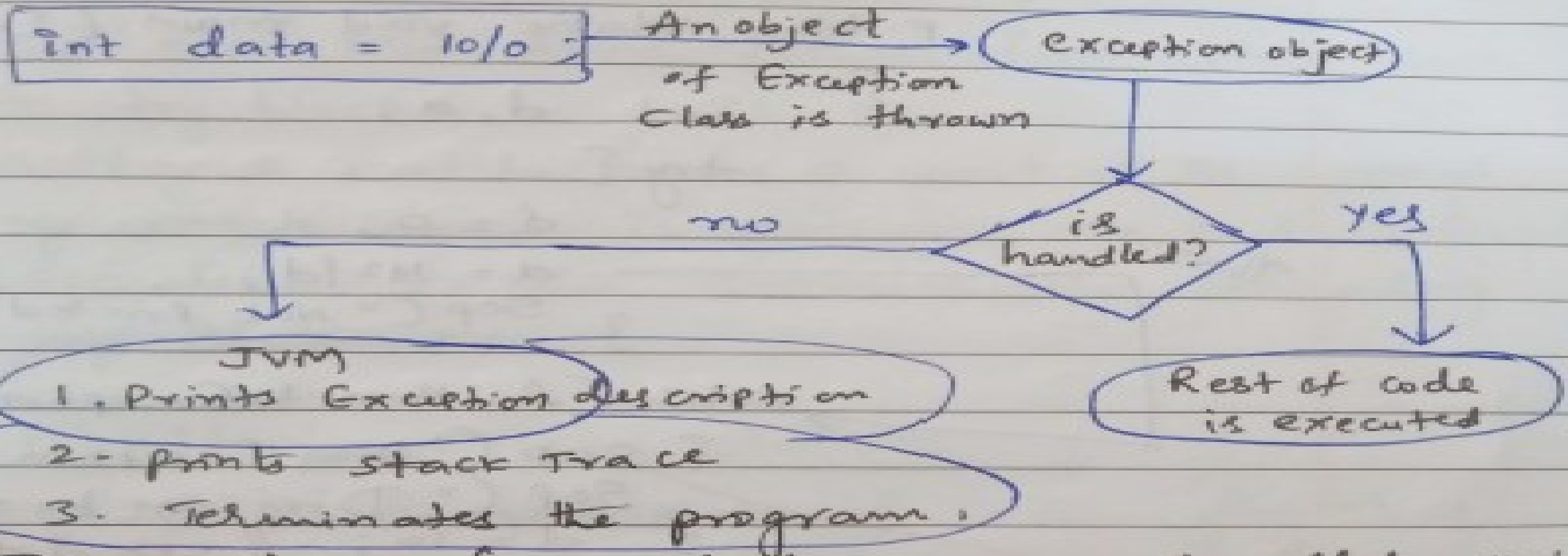
→ Enclose the code that we want to monitor inside a try block.

→ A catch clause that specifies the exception type that we wish to catch (exception handler).

Ex:

ExcepDemo2 class.

The Internal Working of Java try-catch block



→ JVM Checks first, whether exception handled or not

→ If exception is handled by the programer, (catch block) normal flow of application is maintained i.e. rest of the code is executed.

Eg:

```
class ExcepDemo5
{
    public static void main (String args[])
    {
        int d, a;
        try {
            d = 0;
            a = 42/d;
            Sop ("Not printed");
        }
        catch (ArithmeticException e)
        { Sop ("Exception : " + e);
          Sop ("Division by zero");
        }  d = 10;  // set d to other value
        Sop ("After catch statement");
    }
}
```

Thrown

Displays description of exception

o/p : Division by zero
After catch statement

→ Once an Exception is thrown, control transfer from try to catch block.

→ The execution never return from catch block to try block.

→ A try & catch forms a unit. The scope of try catch block is restricted to those statement specified by the immediately preceeding try statement.

(2) Multiple catch clauses

— In a single piece of code (try block) if more than one exception could be raised. Then seperate catch clause is provided for each type of exception

— when an exception is thrown, each stmt is inspected in order, the first one which ever is matched that handler is executed.

— After handling the exception, control transfer to after the try/catch block.

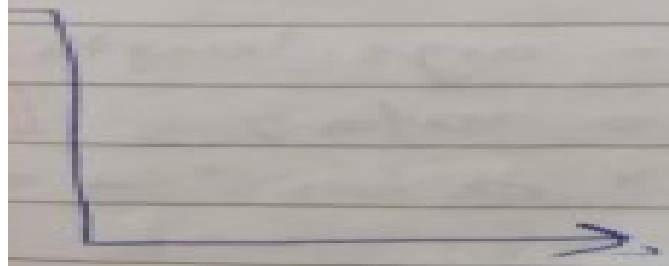
Ex: 0

```
class ExcepDemo7
{
    p s v m (String args[])
    {
        try {
            int a = args.length;
            Sys.out.pr("a = " + a);
            int b = 42/a;
            int c[] = {3, 5, 9, 12};
            c[5] = 99;
        }
        catch(ArithmeticException e)
        {
            Sop("Division by zero");
            Sop(e);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            Sop("Array accun exceeded");
            Sop(e)
        }
        Sop("After try/catch block");
    }
}
```

c:\> java ExcepDem7 2 3 5
↳ ArrayIndexOut of BoundsException

c:\> java ExcepDemo7
↳ Arithmetic Exception.

```
import java.util.*;
public class ExcpDemo6
{
    p s v m (String arg[])
    {
        Scanner sc = new Scanner(System.in
        Sop("Enter no. of subjects");
        int n = sc.nextInt();
        int m[] = new int[n];
        double sum = 0;
    Sop("Enter the marks");
    for(int i=0; i<n; i++)
        {
            m[i] = sc.next();
            sum = sum + m[i];
        }

try {
        double avg = sum/n;
        Sop("avg = " +avg);
    m[10] = 25;
}
```



```
} }
```

Page No.

3. Nested try statements

→ A try statement can be inside the block of another try.

→ If an inner try statement does not have a catch handler, the next try statement's catch handler is inspected for a match.

→ If no catch statement matches, then the Java Run Time System will handle the exception.

Ex:

```
Public class ExcepDemo8
{
    p s v m (String args[])
    {
        try {
            int a = args.length;
            int b = 42/a;          // Ar. Ex
            Sop (" a = " + a);
```

outer

```
try {
    if (a == 1)
        a = a / (a - a);        // Ar. Ex
    if (a == 2)
    {   int c[] = {3, 5, 6};
        c[20] = 99;             // Array. Ex
    }
} catch (ArrayIndexOutOfBoundsException e)
    {   Sop (" Error " + e);
    }
} catch (ArithmeticException e)
    {
        Sop (" Divide by 0" + e);
    }
} // main
} // class.
```

> Java ExcepDemo8 // Ar. Ex

> Java ExcepDemo8 2 // Ar. Ex

> Java ExcepDemo8 2 3 // Array. Ex.

Page No.

Date ___/___/___

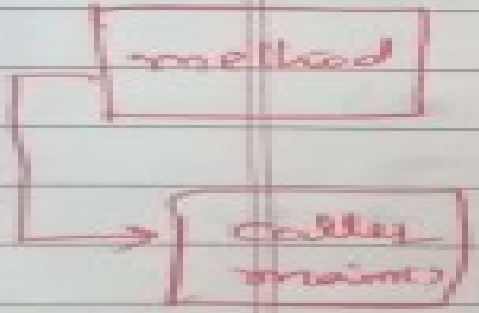
* As of now all are thrown by JavaRuntime System

Nesting with methods —

```
public class ExcepDemo9 {
    static void method (int a)
    {
        try {
            if (a == 1)
                a = a/a-a;
            if (a==2) {
                int c[] = {3,4};
                c[10] = 20;
            }
        }
        catch(AIOOBE e)
        { Sop(e);
        }
    }
    p s vm (String args[])
    {
        try {
            int a = args.length;
            int b = 42/a;
            Sop("a = " + a);
            method(a);
        }
        catch (AE e)
        { Sop(e); }
    }
}
```

Nested try

NOTE: If a method can't provide an handler, it catch (written to) its calling (method).



Page No.

4. throw Statement

A Program can throw an exception explicitly using - throw statement.

throw ThrowableInstance ;

Here the ThrowableInstance must be an object of type Throwable or subclass of Throwable.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

All possible catch blocks are inspected. if no matching handler is found, then the default handler halts the program and prints the stack trace.

Any user defined exception is thrown by using "throw" Statement.

Eg:

```
class ExcepDemo10
{
    static void perform()
    {
        try {
            // String str = null;
            // SOP(str.length());
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            SOP("Exception caught in method");
            throw e; // rethrow the exception
        }
    }
    p s v m(String args[])
    {
        try {
            perform();
        }
        catch(NullPointerException e)
        {
            SOP("Recaught " + e);
        }
    }
}
```

Example

o/p: Exception caught in method
Recaught : Object dsjnfnfnv : demo

→ A exception handler that catches the exception can rethrows it to the outer handler., called Rethrowing exception.

throw new NullPointerException("demo")

new → is used to construct an instance of class.

→ The Exception class will have two Constructors.

① one is no parameter

② one String parameter, here the String describes the description of exception.

→ The msg description is displayed when the object is displayed.

i.e. S.o.p ("Exception" + e);

5. throws clause

If a method is capable of causing exceptions that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against the exception. This can be done by using throws in the method declaration.

* It is not used for Error, RuntimeException and their subclasses.
* All other exceptions that a method can throw must be declared in throws clause. If they are not, a compile-time error will result.
i.e. All checked exceptions.

```
type method_name (Par list) throws Exception list
{
    // method body
}
```

page no.

Eg: ① public static void main (String args[])
throws IOException,
SQLException

{ ≡ }

↓

JVM Handles.

class ThrowsDemo {

static void method() throws
IllegalAccessException

{

S.o.p ("Inside the Throw method");
throw new IllegalAccessException ("demo");

}

ps vm (String args[])

{

try {

method();

}

catch (IllegalAccessException e)

{ SOP ("Caught in main");
SOP(e); }

} }

o/p: Caught in main
java.lang.IllegalAccessException: demo

6. finally Keyword :

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following try/catch block

- finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

- Any time when a method is about to return to the caller from inside a try/catch block, the finally clause is executed just before the method returns.

* If we don't handle an exception, before terminating the program JVM executes finally block (if any)

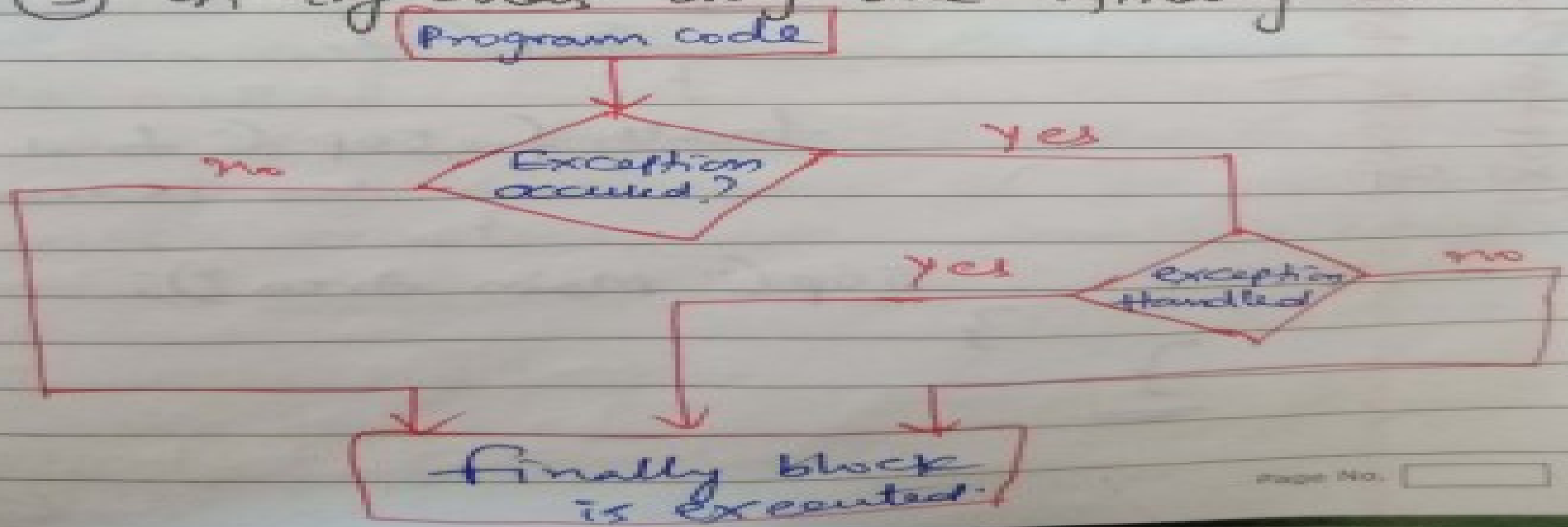
① Java finally block is used to execute important code such as closing connection, file handling etc

② The finally clause is optional. But each try stmt will have a catch or finally block.

③ The finally block is executed whether the exception is handled or not.

④ Used for clean up code.

⑤ A try has only one finally block.



Eg:

```
public class FinallyDemo
{
    P S V M ( String args[])
    {
        try {
            int d = 25/0;
            System.out.println(d);
        }
        catch (ArithmeticException e)
        {
            Sop(e);
        }
        finally { sop (" final block is
                        executed ");
        }
        Sop(" Main Area ");
    }
}
```

Exception handled by finally block is executed.

←

Differences b/n throw & throws.

throw	throws
1. used to throw explicitly our exception	1. used to declare our exception
2. Checked exceptions can not be propagated thru throw stmt	2. checked exceptions can be propagated with throws.
3. throw is followed by our instance	3. throws is followed by class.
4. throw is used within the method	4. throws is used with the method Signature
5 we can not throw multiple exceptions	6. we can declare multiple exceptions.

Creating Own Exception Subclasses (Custom Exceptions).

— If we are creating our own exception, then that is known as custom exception (or) user defined exception.

— Java custom exceptions are used to customize the exception according to user needs.

→ i.e. To create our own exception & message — Any user defined exception must extends Exception class.

Eg:

```
class InvalidAgeException extends
                                Exception
{
    InvalidAgeException (String s)
    {
        super(s);
    }
}
```

↓

```
public String toString()
{
    return "Age Exception";
}
```

```
public class UserException
{
    p s void validate ( int age) throws
    {                          InvalidAgeException
        if (age < 18)
            throw new InvalidAgeException
                                ("Not valid");
        else
            SOP ("Go for Voting");
    }
public static void main (String arg[])
{
    try {
        validate (20);
        validate (17);
    }
catch (Exception m)
    {
        SOP ("Exception caught in main");
    } SOP (m); // description of exception
} } SOP ("Rest of code");
```

Java's Built-in Exceptions

Date ___/___/___ Saathi

The package "java.lang" defines to several exception classes.

The Unchecked exceptions defined in java.lang are the

ArithmeticException → Arithmetic Error, such as division by zero

ArrayIndexOutOfBoundsException → Array index is out of bounds.

ArrayStoreException → Assignment to an array element of an incompatible type.

ClassCastException → Invalid cast of class

IllegalArgumentException → Illegal argument used to invoke a method

IndexOutOfBoundsException → some type of index is out-of-bounds.

NegativeArraySizeException → Array created with a negative size.

NullPointerException — Invalid use of null reference

NumberformatException — Conversion of a string to numeric format.

SecurityException — Attempt to violate security

The Checked Exceptions defined in java.lang, must be included in "throws" list.

ClassNotFoundException — class not found.

IOException — I/o Read/write error

IllegalAccessException — Access to a class is denied.

InstantiationException — Attempts to create an object of abstract class/interface

InterruptedException — One thread has been interrupted by another thread.

NoSuchField Exception — A requested field does not exist.

NoSuchMethod Exception — A requested method does not exist.

Page No.

```
//Without Exception Handling
public class ExcepDemo1
{
public static void main(String args[])
{
int y=50;
int x=y/0;//may throw exception
System.out.println("rest of the code...");
System.out.println("Welcome to Exception
Handling");
}
}
```

```
//With Exception Handling
public class ExcepDemo2
{
public static void main(String args[])
{
try
{
int y=50;
int x=y/0;//may throw exception
System.out.println("Hi hello");
}
catch (ArithmeticException e)
{
System.out.println(e); //exception object
description
System.out.println("Division Error - Should
not be zero in denominator");
}
System.out.println("rest of the code...");
```

```
//Uncaught Exception - StackTrace
public class ExcepDemo4
{
public static void method()
{
int d=0;
int a = 10/d;
}
public static void main(String[] args)
{
System.out.println("Hello World!");
method();
}
}

//Program to use try-catch keywords
class ExcepDemo5
{
public static void main(String args[])
{
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be
printed.");
}
catch (ArithmeticException e)
{
// catch divide-by-zero error
System.out.println("Division by zero.");
System.out.println("Exception is " +e);
//displaying exception type
d = 20; //set d to other than zero
}
System.out.println("After the cathch
statement ");
System.out.println("d value is " +d);
}
}
```

```
//multiple catch blocks
import java.util.*;
public class ExcepDemo6
{
public static void main(String args[])
{
Scanner sc = new Scanner(System.in);
System.out.println("Enter the no of
subjects");
int n = sc.nextInt();
System.out.println("The no of Subjects is "
+n);
int m[] = new int[n];
double sum=0;
System.out.println("Enter the marks");
for(int i=0;i<n;i++)
m[i] = sc.nextInt();
for(int i =0;i<n ; i++)
sum=sum+m[i];
try
{ //int y = 25/0;
double avg = sum/n;
System.out.println("The average of the
student is "+avg);
m[10] =25;
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Accessing out of array");
System.out.println("Exception is " +e);
}
catch (ArithmeticException e)
{
System.out.println("No of Subjects must not
be Zero");
System.out.println(e);
}
System.out.println("after the catch block");
}
}
```

```
// Demonstrate multiple catch statements.
class ExcepDemo7
{
public static void main(String args[])
{
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

```
// An example of nested try statements.
public class ExcepDemo8
{
public static void main(String args[])
{
try {
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try {
if(a==1) a = a/(a-a); // division by zero
if(a==2)
{
int c[] = { 1,2,3 };
c[42] = 99;
// generate an out-of-bounds exception
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-
bounds: " + e);
}
} //outer try
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e+
"outer");
}
}
}

//java ExcepDemo8 -> Ar.Excp
//java ExcepDemo8 9 -> Arth. Excp
//java ExcepDemo8 12 13 -> array excp.
```

```
// Demonstrate throw.
class ExcepDemo10
{
static void perform()

try {
// String str=null;
// System.out.println(str.length());
throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside
perform."+e );
// System.out.println(e);
throw e; // rethrow the exception
} }
public static void main(String args[])
{
try { perform(); }
catch(NullPointerException e)
{
System.out.println("Recaught: " + e+
"main");
} }
```

```
// trows clause demo
class ThrowsDemo
{
static void method() throws
IllegalAccessException
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
try {
method();
}
catch (IllegalAccessException e)
{
System.out.println("Caught " + e);
```

```
//finally demonstration
public class FinallyDemo
{
public static void main(String args[])
{
try{
perform();
}
catch (ArithmeticException ae)
{
System.out.println("Exception caught in
main");
System.out.println(ae);
}
finally
{
System.out.println("Finally executed in
main");
}
}
```

```
public static void perform()
{
try{
int x= 25;
int d = x/0;
}
catch(ArrayIndexOutOfBoundsException ae)
{
System.out.println(ae);
}
finally
{
System.out.println("Finally executed in
method");
}
}
}
```

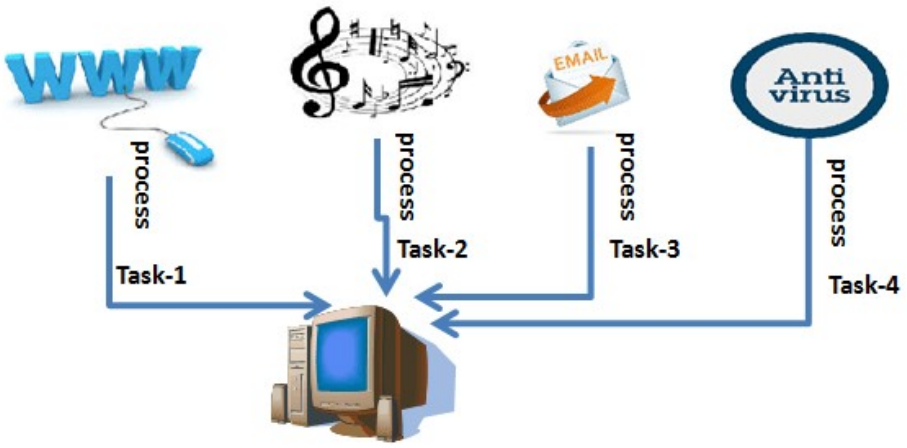
```
//Userdefined Exceptions
class InvalidAgeException extends Exception
{
InvalidAgeException(String s)
{
super(s);
}

public String toString()
{
return "Age Exception";
}
}

class UserExcepDemo
{
static void validate(int age)throws
InvalidAgeException
{
if(age<18)
throw new InvalidAgeException(" not valid ");
else
System.out.println(" welcome to vote "
+age);
}

public static void main(String args[])
{
try{
validate(23);
validate(17);
}
catch(InvalidAgeException m)
{
System.out.println("Exception occured: ");
System.out.println( m); // Description of
Message
}
System.out.println("rest of the code...");
```

1. Understand Multi Tasking with Processes and Threads.

Multi Tasking

 Deal with more than one task at the same time.

 Execute more than one program or task simultaneously.

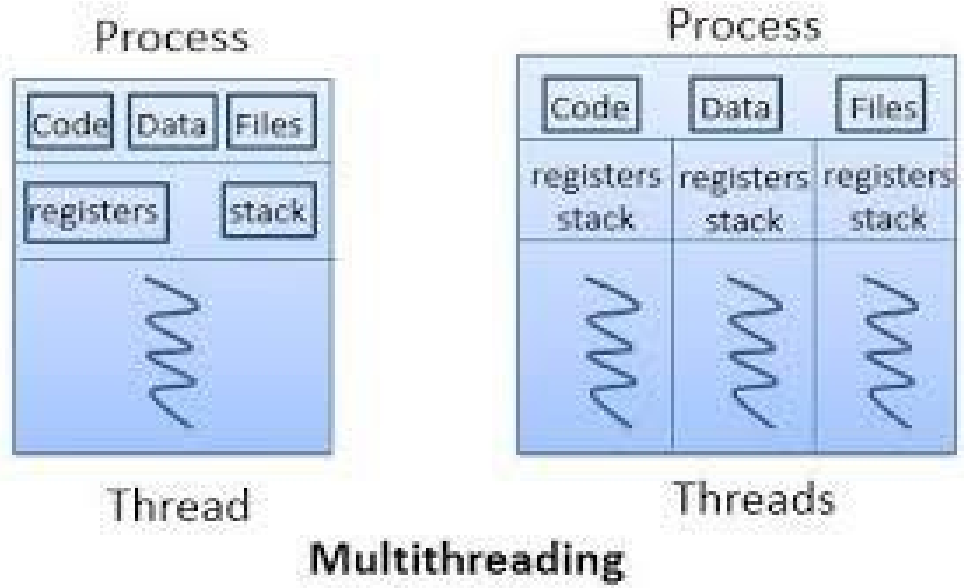


Multitasking On a Desktop System

Multi tasking is implemented in computer system using two ways.

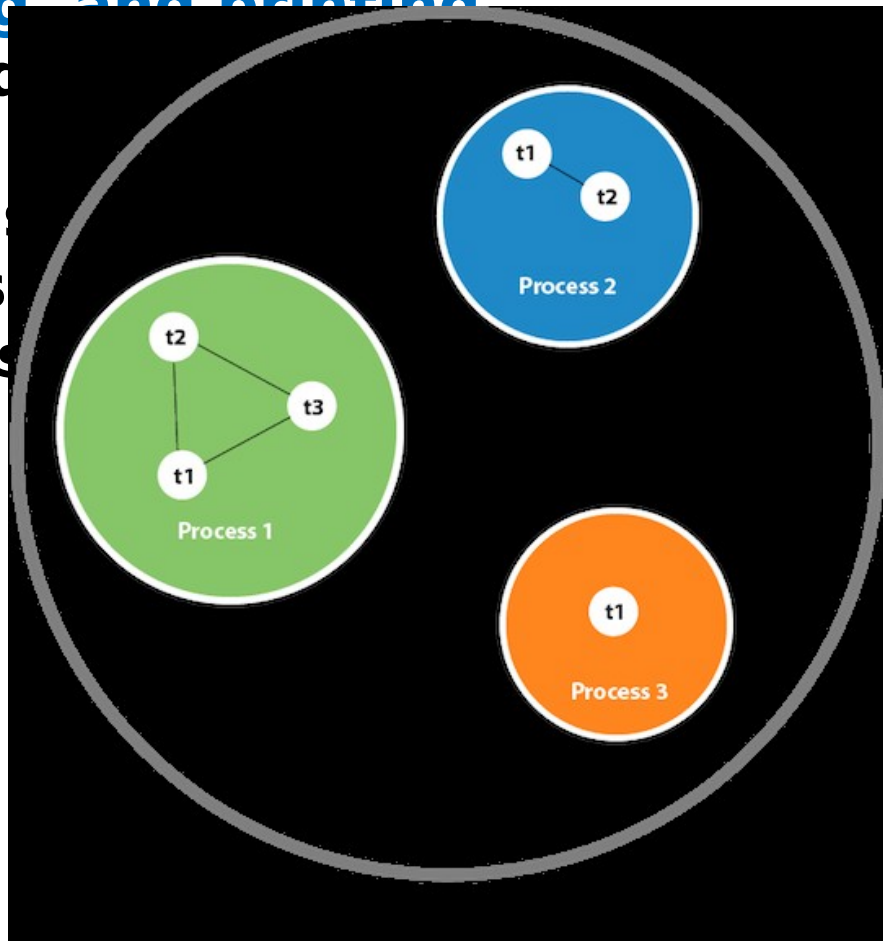
1. Process Based Multi Tasking

- A process is a program under execution
- Allows the computer to perform two or more programs concurrently.
- Called Process based Multi Tasking
- Run a Java Compiler and Text Editor at san
- Each process has its own address space in
- Process is a heavy weight processing unit
- Cost of communication between processes
- Switching from one process to another is c
- Inter process communication is expensive
- It is in the control of Operating System.
- It is not in the control of JAVA or JVM
- Process based multitasking requires more overhead.
- Taking control of CPU is difficult.



2. Thread Based Multi Tasking

- A Thread is the smallest unit of dispatch able code, having its own execution path.
- A single program can perform two or more tasks simultaneously.
- Called Thread based Multi Tasking
- Editing on a text file, checking for spelling, and printing
- All the threads will share the common add
- Thread is a light weight processing unit
- Cost of communication between threads i
- Switching from one thread to another is s
- Inter thread communication is less expens
- Threads are in the control of Java or JVM.
- Taking control of CPU is easy.



Multi Threading

- Java Provides built in support for multi threaded programming.
- A Multi threaded program contains two or more parts that can run concurrently. Each part of such a program is called a **Thread.**
- Each thread defines a separate **execution path**.
- Multi Threading is a specialized form of **multi tasking**.
- A thread is a smallest unit of processing, it has separate execution path.
- Thread are **independent,** if exception occurs in one thread it does not affect other threads.
- Threads share **common** memory area
- One process can have multiple threads; the operating system can have

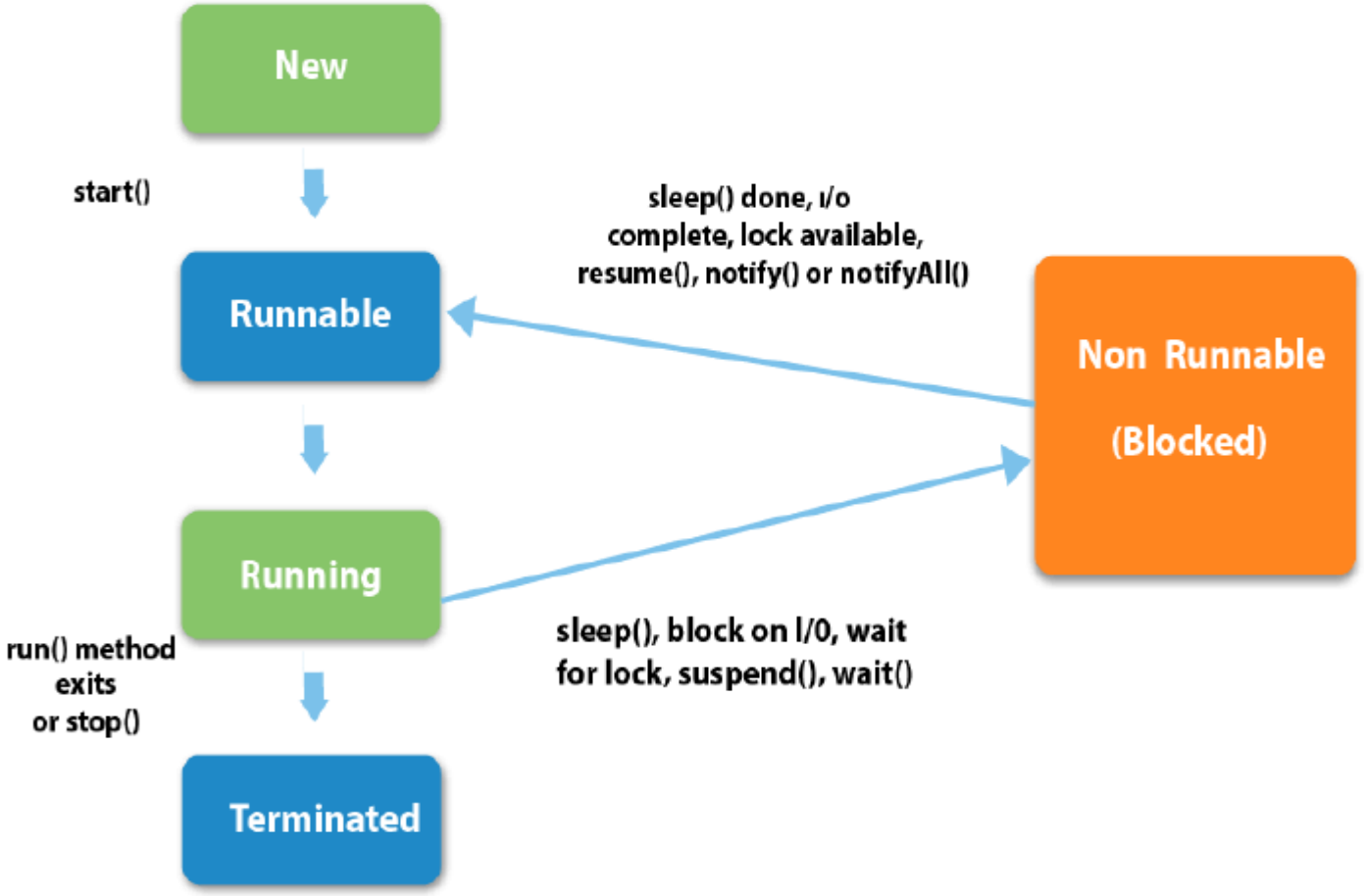
Advantages of MultiThreading

- Allows to write very efficient program that makes maximum use of CPU.(Idle time can be kept minimum).
- Eg: During I/O operation , CPU switch to another thread.
- Threads are independent
- Throughput – Amount of work done in unit time increase.

Applications of Java MultiThreading

Thread Life Cycle

- The life cycle of a thread in Java is controlled by JVM.
- During the execution of thread, it is in any of the following five states.
- Hence Java thread life cycle is defined in five states.



New -> The thread is in new state when the instance of thread class is created but before the invocation of **start()** method.
Runnable->The thread is in runnable state after the invocation **start()** method, but the thread scheduler has not selected it to be the running thread. Any number of threads exists in this state.
Running-> The thread is in running state when the thread scheduler selects a thread for execution.
Non-Runnable -> this is the state when the thread is still alive but is not eligible currently to run. **(Blocked)**
Terminated -> A thread is in terminated or dead state when its **run()** method exits.

Thread class

- Java provides **Thread** class to achieve thread programming.
- Thread class provides constructors and methods to create and perform operations on a thread.
- **Thread** class extends **Object** class and implements **Runnable** interface.

Create Threads using Thread class and Runnable Interface

Creating Threads

In Java a Thread is created by using two methods

1. Implementing Runnable Interface

2. Extending Thread class

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- **Thread()**
- **Thread(String name)**
- **Thread(Runnable r)**
- **Thread(Runnable r, String name)**

Creating Threads

In Java a Thread is created by using two methods

1. Implementing Runnable Interface

2. Extending Thread class

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- **Thread()**
- **Thread(String name)**
- **Thread(Runnable r)**
- **Thread(Runnable r, String name)**

4. public int setPriority(int priority): Used to set or change the priority of the thread.
5. public String getName(): returns the name of the thread.
6. public void setName(String name): Used to set or change the name of the thread.
7. public static Thread currentThread(): returns the reference of currently executing thread.
8. public int getId(): returns the id of the thread.
9. Public static void sleep(long miliseconds): Causes the currently executing thread to suspend (temporarily cease execution) for the specified number of milliseconds.
public static void sleep(long miliseconds)
public static void sleep(long milliseconds, int nanosecs)
10. public void join(): waits for a thread to die/terminate.
11. public void join(long miliseconds): waits for a thread to die or terminate for the specified miliseconds.
12. public boolean isAlive(): to check the thread is still running. Returns true if the thread is still running.
13. public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

***The Thread class methods throws InterruptedException

Main thread

```
import java.lang.*;
public class MainThread
{
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("The current thread is
"+t);
// [Thread Name ,
Priority ,ThreadGroupName]
System.out.println("The name of Thread
is "+ t.getName());
t.setName("Demo Thread");
System.out.println("The name of Thread
is "+ t.getName());
System.out.println("The priority of
Thread is "+ t.getPriority());
try{
for(int i=1; i<=5;i++)
{
System.out.println(i);
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
System.out.println(e);
}
}
}
```

- When a Java program executes, the first thread starts is main thread.
- The Main thread is important for
  - It is the thread from which other child threads re created.
  - It must be the last thread to finish execution, because it performs various shutdown actions.
- The main thread is created automatically when a program started.

Creating Thread – Extending Thread Class

Step-1: **Create a new class that extends Thread, and then create an instance of that class.**

Step-2: **The extending class must override the run () method, which is the entry point of the new thread created. The actual code for the thread to execute is provided. Once the run () method completes, the thread will die and terminate.**

Step-3: **Calls start () method to begin execution of new thread.**

Step-4: **Invoke the Thread class constructor using super keyword if necessary.**

```
class MyThread extends Thread
{
public MyThread(String name)
{
super(name);
}
public void run()
{
System.out.println("MultiThreading");
System.out.println("Child Thread -> " +
Thread.currentThread());
}
}
public class TheadDemo1 {
public static void main(String[] args) {
System.out.println("Main Thread");
System.out.println(Thread.currentThread());
MyThread t1 = new MyThread("One");
t1.start();
System.out.println("The child Thread -> "+t1);
```

Creating Thread – Implementing Runnable Interface

Step-1: To create a new Thread, the class must implements Runnable interface.

Step-2: Provide implementation for the only one method run (), which is the entry point of newly created thread. The actual code for the thread to execute is provided. Once the run () method completes, the thread will die and terminate.

Step-3: Instantiate an object of type Thread within the newly created thread class.

Thread(Runnable obj, String name)

Step-4: call the start () method explicitly to start the thread. It makes a call to run() method to start the execution.

```
class DemoThread implements Runnable
{
 Thread t;
 public DemoThread()
 {
 t= new Thread(this , "One");
 System.out.println(t);
 t.start();
 }
 public void run()
 {
 System.out.println("MultiThreading ");
 } }
 public class ThreadDemo2 {
 public static void main(String[] args) {
 System.out.println(Thread.currentThread());
 DemoThread d = new
```

Creating Mutliple Threads using Thread class

A program can spawn as many threads as needed.

```
//Multithreading By Extending Thread class
class NewThread extends Thread
{
NewThread(String threadname) {
super(threadname);
System.out.println("Child Thred -> " +this );
//start(); // Start the thread
}
public void run(){
try {
for(int i = 5; i > 0; i--){
System.out.println(getName ()+ ": " + i);
sleep(500);
}
}
System.out.println(getName() + " Completed");
}
}
class MultiThread1
{
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("Name is --->" +t.getName());
System.out.println("Main ---> " +t);
NewThread t1 = new NewThread("One");
//or new NewThread("ONE");
NewThread t2= new NewThread("Two");
NewThread t3 = new
```

```
try {
// wait for other threads to end -give long time
for(int n=1;n<=5;n++){
System.out.println("Main Thread->" +n);
Thread.sleep(500);
}
catch (InterruptedException e){
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread Completed");
}
}
```

Creating Mutliple Threads using Runnable Interface

```
class NewThread implements
Runnable {
Thread t;
NewThread() {
// Create a new thread
t = new Thread(this, "Child
Thread");
System.out.println("Child thread: "
+ t);
System.out.println("The thread
name is "+t.getName());
t.start(); // Start the thread
}
// This is the entry point for the child
thread.
public void run() {
try{
for(int i = 1; i <=10; i++){
System.out.println("Child Thread: -
catch (InterruptedException e){
System.out.println("Child interrupted.");
} } }
public class MultiThread2{
public static void main(String args[ ]
) {
//Thread t = Thread.currentThread();
//System.out.println("The thread
name is " +t.getName());
// create a new thread-child thread
NewThread t1 = new NewThread();
NewThread t2 = new NewThread();
NewThread t3 = new NewThread();
try{
```

Creating Threads of Multiple Classes

```
import java.lang.*;
class FactThread extends Thread
{
FactThread(String name) {
super(name);
System.out.println("The child is " +this);
}
public void run() {
System.out.println("Enter a number");
Scanner sc = new Scanner(System.in);
int n= sc.nextInt();
int f=1;
try{
for(int i=1 ; i<=n ; i++){
f= f*i;
System.out.println("Doing --wiat");
Thread.sleep(1000);
}
}
catch(InterruptedException ie){
System.out.println("Exception");
}
System.out.println("The factorial value is
" +f);
}
}
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread get more CPU time than lower-priority thread.
- A higher-priority thread can also pre-empt a lower-priority one. For instance, when a lower-priority thread is running and a higher priority thread resumes (from sleeping or waiting on I/O; for example), it will pre-empt the lower-priority thread.

Thread class defines the method setPriority() to assign priority to a thread.

final void setPriority(int level)

The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively.

To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.

public static final int MIN_PRIORITY (1)

public static final int MAX_PRIORITY (10)

public static final int NORM_PRIORITY (5)

To obtain the current priority setting by calling the getPriority( ) method of Thread class.

```
class NewThread extends Thread {
NewThread(String name){
super (name);
}
public void run(){
System.out.println("Hello -->"+
getName() + "->"+getPriority());
}
}
public class PriorityDemo {
public static void main(String args[]) {
// System.out.println("MainThread
Priority
is:"+Thread.currentThread().getPrio
rity());
Thread t = Thread.currentThread();
System.out.println("The main thread
priority is " + t.getPriority());
NewThread t1 = new NewThread("One");
NewThread t2 = new NewThread("Two");
NewThread t3 = new
NewThread("Three");
t1.setPriority(Thread.MIN_PRIORITY)
;
t2.setPriority(Thread.MAX_PRIORITY
);
t3.setPriority(5);
t1.start();
t2.start();
t3.start();
}
}
```

Synchronizing Threads

- **When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.**
- **Java provides unique, language-level support for it.**
- **Synchronization allows only one thread to access a shared resource.**
- The advantages of Synchronization is
  - **To prevent thread interferences.**
  - To prevent inconsistency of data.

Thread Synchronization can be of two forms

1. Mutual Exclusion

2. Inter Thread communication.

1. Mutual Exclusion

Keeps threads from interfering with one another while sharing data.
Allows only one thread to access shared data

Mutual exclusive threads can be implemented using

i) Synchronized Method

ii) Synchronized Block

i) Synchronized Method

A method defines as synchronized, and then the method is a synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires lock for that object and releases it when the thread completes its task.

```
//synchronized Method - for Synchronization
class Table
{
synchronized void printTable(int n)
{
try{
for(int i=1;i<=10;i++) {
System.out.println(n*i);
Thread.sleep(1000);
}
}
catch(InterruptedException e) {
System.out.println(e);
}
}
}
```

```
class MyThread1 extends Thread {
Table t;
MyThread1(Table t) {
this.t=t;
}
public void run() {
t.printTable(8);
}
}
class MyThread2 extends Thread {
Table t;
MyThread2(Table t) {
this.t=t;
}
public void run() {
t.printTable(9);
}
}
```

```
public class SyncDemo1 {
public static void main(String args[]) {
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

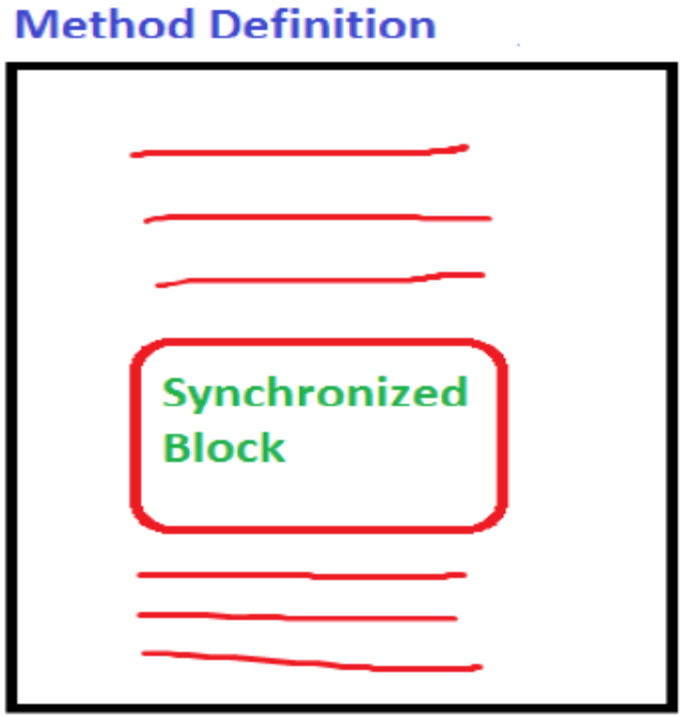
ii) Synchronized Block

Synchronized block can be used to perform synchronization on any specific resource of a method.
Synchronized block is used to lock an object for any shared resource.
Scope of Synchronized block is smaller than the method.

```
//Thread Synchronization using - synchronized block
class Table {
void printTable(int n)
{
System.out.println("*****Hello -not synched");
//synchronized block
synchronized(this) {
for(int i=1;i<=10;i++) {
System.out.println(n*i);
try{
Thread.sleep(1000);
}
catch(InterruptedException e) {
System.out.println(e);
}
}
} //end of sync block
System.out.println("Hello -not synched");
} //end of the method
}
```

```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run() {
t.printTable(5);
}
}
class MyThread2 extends Thread
{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run() {
t.printTable(100);
}
}
```

```
public class SyncDemo2 {
public static void main(String args[])
{
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```



The form of synchronized block is

```
synchronized(object) {
//Statements to be synchronized
}
```

The form of synchronized method is

```
class Table
{
synchronized type
methodName(arguments)
{
//method body for Synchronization
}
}
```

InterThread Communication

Inter thread communication or cooperation is allowing synchronized threads to communicate with each other.
Interthread communication is a mechanism in which a thread is paused running in its critical section (synchronized block) and another thread is allowed to enter into the same section.
Eg: Producer-Consumer Problem.
Java Provides Interthread communication by using three methods of Object class.
1. wait() 2. notify() 3. notifyAll()

All the three methods must be called only from within a synchronized context.
All these methods are related to lock and the object has a lock.
1. wait() – Tells the calling thread to given up the monitor(lock) and go to sleep until some other thread enters the same monitor and calls notify().

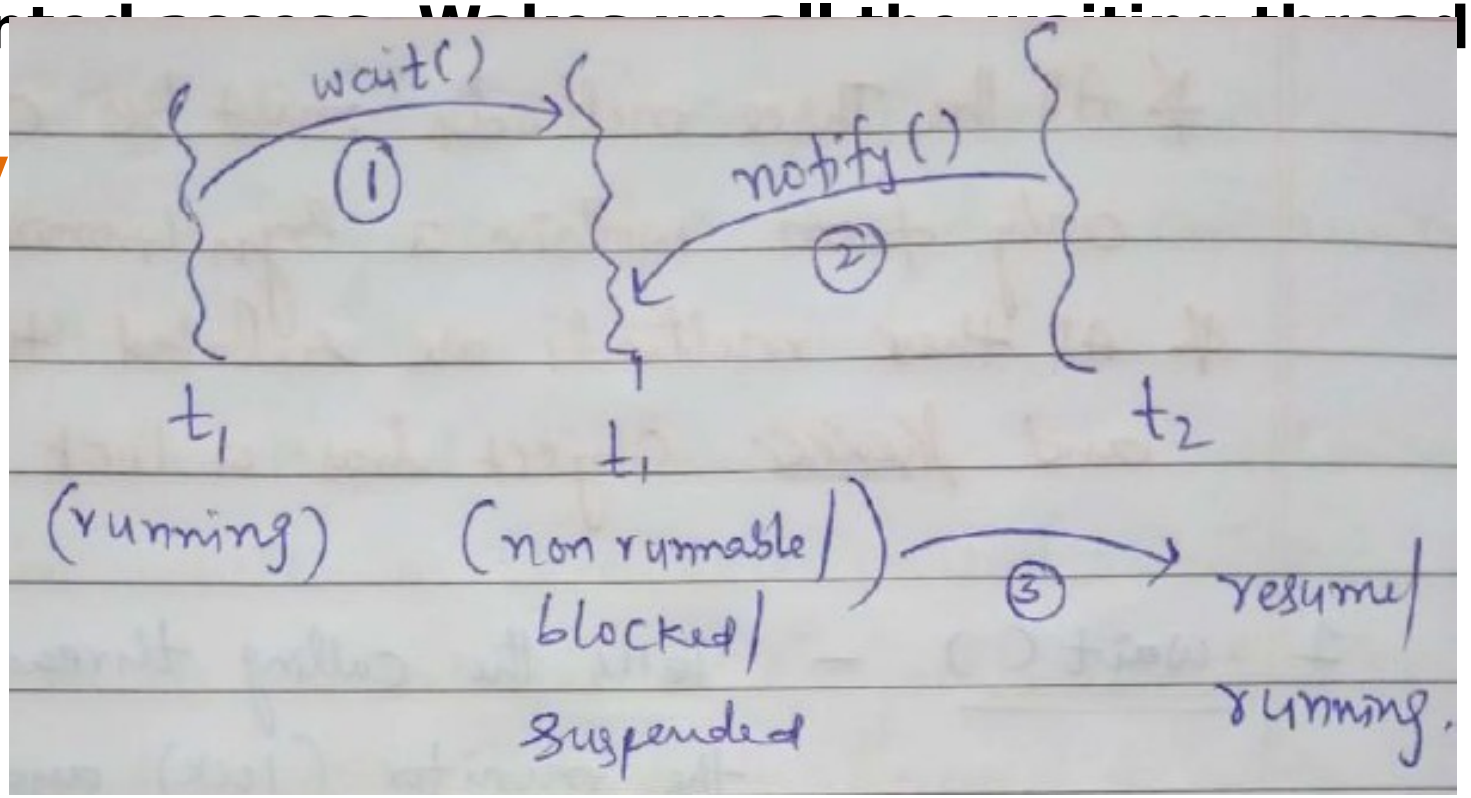
public final void wait() throws InterruptedException
public final void wait(long time) throws InterruptedException
Causes current thread to wait until another thread invokes the notify()

2. notify() – wakes up a single thread that is waiting on this object's monitor. If many threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

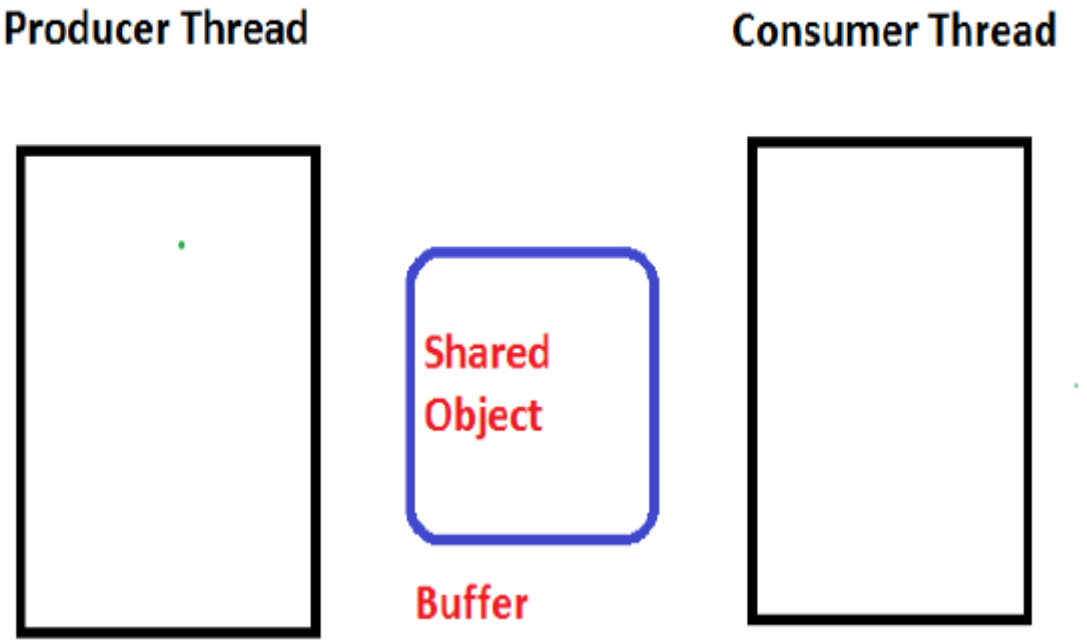
public final void notify()

3. notifyAll()- wakes up all thread called wait() on the same object. One of the thread will be gran ls.

public final void notify



Producer Consumer Problem for Interthread Communication



Producer Thread- Produce items to Buffer (Add Items)
Consumer Thread- Consume items from Buffer (Removes Items)
The two conditions for Producer – Consumer Problem is
1. Producer cannot add an item into a buffer if it is full
2. Consumer cannot consume an item if it is empty.

If no communication, these two conditions are not satisfied then the CPU is always in polling (loop). To Save CPU time in Java Interthread communication is used.

```
//Producer-Consumer problem --->
Inter Thread Communication.
class Buffer
{
int item;
boolean produced = false;
synchronized void produce(int x)
{
if(produced)
{
try{
wait();
}
catch(InterruptedException ie)
{
System.out.println("Exception Caught");
}
}
item =x;
System.out.println("Producer - Produced-->" +item);
produced =true;
notify();
}
synchronized int consume()
{
if(!produced)
{
try{
wait();
}
catch(InterruptedException ie)
{
```

```
 System.out.println("Consumer -
Consumed " +item);
produced = false;
notify();
return item;
}
}

class Producer extends Thread
{
Buffer b;
Producer( Buffer b)
{
this.b = b;
start();
}
public void run()
{
b.produce(10);
b.produce(20);
b.produce(30);
b.produce(40);
b.produce(50);
}
}
```

```
class Consumer extends Thread
{
Buffer b;
Consumer(Buffer b)
{
this.b = b;
start();
}
public void run()
{
b.consume();
b.consume();
b.consume();
b.consume();
// b.consume();
// b.consume();
// b.consume();
}
}
```

```
public class PCDemo
{
public static void main(String args[])
{
Buffer b = new Buffer(); //Synchronized
Object
Producer p = new Producer(b);
Consumer c = new Consumer(b);
}
}
```